Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah …

1 EINLEITUNG

Blah …

2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Blah …

3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

Blah …

4 MODERNE ENTERPRISE-ARCHITEKTUREN

4.1 Microkernel Architecture

Die Microkernel Architecture ist ein Architekturmuster, was sich durch erweiterbarkeit, flexibilität und vor allem isolation der Funktionalitäten auszeichnet. Wie in Abbildung 1 dargestellt, enthält ein Mikrokernel zwei wesentliche Komponenten: Den Kern der Anwendung,der die wichtigsten grundlegenden Funktionalitäten bereitstellt und Module oder auch Plugins, die diesen Kern um Features und weitere Business Logik erweitern [Richards 2015].

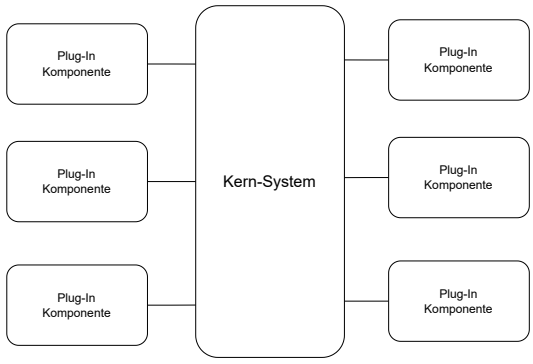

Abb. 1. Aufbau einer Microkernel Architecture

Der Kern der Anwendung implementiert dabei meist nur die minimalste Funktionalität, um die Anwendung oder das System lauffähig zu machen. Alle weiteren Funktionalitäten werden in Modulen implementiert, die auf den Kern aufbauen. Module sind meist unabhängig voneinander aufgebaut, es ist kann jedoch auch vorkommen, dass manche Module von anderen abhängig sind. Dabei ist aber die best practice zu beachten, Kommunikation zwischen einzelnen Modulen so gering wie möglich zu halten, um Probleme durch Abhängigkeiten zu vermeiden. Dadurch sind Module untereinander lose gekoppelt und können unabhängig voneinander entwickelt, getestet und deployed werden.

Die Plugins können über verschiedene Wege mit dem Kern verbunden werden. Eine genaue Spezifikation zum Verbinden der Plugins mit dem Kern gibt es aber laut Architekturschema nicht, diese Entscheidung ist dem Entwickler überlassen und entsprechend der Anforderungen und Anwendungsumgebung zu treffen. Beispielsweise könnte die Verbindung über Web-Services, Messaging oder am einfachsten über direkte Objekt-Instanziierung innerhalb der gleichen Anwendung stattfinden.

Zwischen Plugins und Kern werden Contracts definiert, die die Kommunikation zwischen den beiden Komponenten regeln. Diese Standard Contracts können in Form von Interfaces, Klassen oder auch Datenstrukturen definiert werden. Sollten Third Party Plugins zum Einsatz kommen, müssen sich diese an die definierten Contracts halten, um mit dem Kern kommunizieren zu können. Alternativ können auch Adapter verwendet werden, um bestehende Plugins an den Kern bzw. die Contracts anzupassen.

Ein Problem, was sich durch diesen Aufbau ergibt, ist dass der Kern jederzeit wissen muss, welche Plugins aktuell verfügbar sind und wie diese angesprochen werden können. Um dieses Problem zu lösen, kann eine Plugin-Registry verwendet werden. Diese Registry enthält alle aktuell verfügbaren Plugins und die dazugehörigen relevanten Informationen wie z.B. Name des Service, Data Contracts, Verbindungsdetails, etc. Der Kern der Anwendung kann dann zur Laufzeit auf diese Registry zugreifen und Plugins dynamisch laden.

Microkernel Architekturen können auch in anderen Architekturmustern eingebettet sein, falls es nicht möglich sein sollte die gesamte Software in diesem Architekturmuster aufzubauen. Vor allem Teile von Anwendungen, die stark erweiterbar sein müssen, eignen sich gut für die Verwendung der Microkernel Pattern.

Ein klarer Vorteil dieses Architekturmusters ist die schnelle Reaktionsfähigkeit auf äußere Änderungen, da Anpassungen größtenteils nur in den isolierten Modulen vorgenommen werden. Der Kern der Anwendung ist in den meisten Fällen schnell stabil und benötigt selten im Laufe der Entwicklung weitere Angleichungen. Geänderte Module können je nach Implementierung auch zur Laufzeit geladen bzw. hinzugefügt werden, was mögliche Downtime von Software während des Entwicklungsprozesses minimiert.

Auch profitiert die Testbarkeit der Software durch die lose Kopplung der Module. Jedes Modul kann unabhängig voneinander getestet werden und fehlende Module durch Mocks oder Stubs ersetzt werden, wodurch sich während der Entwicklung auf einzelne Funktionalitäten konzentriert werden kann.

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

4.2 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier „*An introduction to Software Architecture*“ auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Reagiert auf publizierte Events
- Vermittler (englisch *Mediator*): Liegt zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als ein Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 2 stellt diesen Vertrag dar.

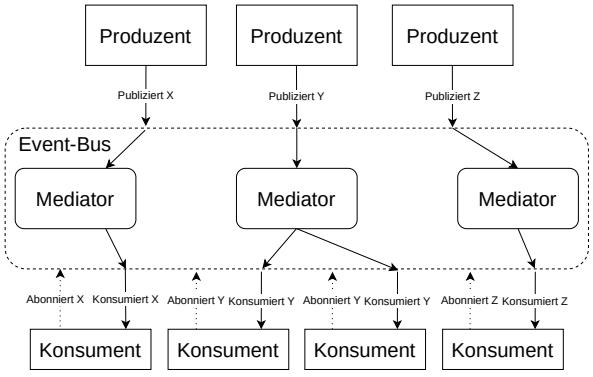

Abb. 2. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht agilen Entwickler-Teams durch klar abgegrenzte Features einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung maximale Skalierung. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Die Agilität der Architektur kann weiter erhöht werden, indem der event-basierte Aspekt mit weiteren agilen Strukturen wie Microservices oder cloud-nativen Serverless-Functions kombiniert wird. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

6 DISKUSSION

7 ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015. *Software Architecture Patterns.* O'Reilly Media, Inc. value pages.

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

A ANHANG 1

A.1 Übungsaufgaben

Blah …

B ANHANG 2

Blah …